TEACHING SEMAPHORES USING . . . SEMAPHORES *

Robert Marmorstein
Longwood University
Ruffner 329
201 High Street
Farmville, VA 23909
(434)395-2185
marmorsteinrm@longwood.edu

ABSTRACT

The word "semaphore" describes both a structure used in computing to manage concurrency and a signal used by trains and naval vessels to transmit information and prevent collisions. In this paper, we present a series of activities which help students build intuition about concurrency by managing "railway semaphores" using an open source train simulation game.

The increasing importance of multi-core computers and computing clusters makes concurrency an especially important topic in both operating systems and systems programming courses. Students often struggle to understand semaphores and concurrency problems. One reason for this is that these subjects are often presented as abstract mathematical objects which are hard to visualize. Using simulation to make these ideas more concrete can improve understanding and increase engagement in the course.

INTRODUCTION

The ability to manage concurrency using semaphores is one of the most important topics of an upper-level operating systems or systems programming course, but it can be very difficult to teach. Concurrency problems, such as race conditions and deadlocks, are abstract concepts that can be difficult for a student to visualize. They can arise in many different situations from file locking to network communication.

Concurrency problems have been a focus of upper-level systems programming and operating systems classes since the early days of computer science[4]. They have become increasingly important as multi-core computers have become standard, not just for servers, but in consumer systems and embedded devices. Concurrency issues arise whenever a multi-threaded program accesses a shared resource, such as a file or a variable in global memory, or when a program can be interrupted to allow some other code to run. This is especially common for high-performance web applications, but is also important in graphical interface design, data storage, and other areas where threading can improve performance or interactivity.

As Akimoto and Cheng[1] point out, teaching concurrency has some unique challenges. Not only is the material particularly difficult for students, but the complexity of the topic makes it difficult to develop projects which engage students and are appropriate for a single semester course. This problem is further complicated by the fact that concurrency problems which seem similar to students often have subtle differences that require very different approaches to their solution. For example, the producer/consumer problem and the readers/writers problem both have two agents which communicate using a shared data structure, but the readers/writers problem typically requires multiple mutual exclusion locks and significantly more sophisticated logic than the producer/consumer problem. These difficulties have prompted several creative approaches to introducing these topics.

Possibly the most common approach is to develop a programming language, library, or framework which allows students to easily use semaphores and monitors in short projects. This approach is used by Pascal-FC[4], BACI[5], and a custom programming package[8] used at Trinity College. The Trinity package is particularly interesting in that it provides specialized debugging and tracing tools that allow students to visualize both threads and synchronization constructs. Many of these short projects involve simple games. The author of [15] uses a similar technique to teach a related topic, interprocess communication mechanisms, using two simple strategy games.

Slightly longer projects, which involve implementing a significant component of an operating system, are also popular. In [7], the authors use a project which requires students to implement large portions of an operating system in assembly language. Other popular projects, such as NachOS[6], PintOS[14], and OS/161[9], allow students to use higher-level languages to apply synchronization techniques to design of major O.S. components.

A second approach is to provide tools for visualizing concurrent interactions in multi-threaded C or Java programs. The authors of [2] and [3] take this approach. Both of these tools provide the student with a graphical depiction of the execution of a group of threads and provide single-stepping capability. This approach supports the use of familiar languages in class projects so that the student does not need to learn new language features to complete the activities. The authors of [10] combined this approach with a project-driven curriculum. Their “Simple OS” project requires students to implement significant parts of an operating system, but provides a visualization framework that aids students in comprehending and debugging synchronization constructs.

A third approach is to motivate concurrency learning through the use of single or multi-player games. In [1], Akimoto and Cheng describe a graphical game they have developed which uses interactive robots to illustrate concurrency principles.

In [11], Kolikant et al. present a particularly intriguing study of the reasons students become confused in operating systems classes. Using anthropological theory, they found that students frequently form a flawed mental model of the behavior of semaphores. They suggest giving assignments that strongly reinforce the principles and behavior of concurrency constructs before assignments that require the students to use them in problem solving. They also emphasize the importance of assignments that build comprehension over assignments that teach students to simply memorize usage patterns.

These experiences strongly suggest that one of the best ways to introduce concurrency is to use games and other interactive projects to give students a stronger intuition about the semantics of semaphores. This paper presents a set of hands-on activities which use an open source train simulation game, OpenTTD, to visually represent concurrency problems. Each activity introduces one application of semaphores and allows the students to interact with the system by placing "real" semaphores along a train track carefully constructed to simulate a computing problem (such as a race condition). The activities can be downloaded from my website at http://marmorstein.org/~robert/trains.html and consist of an OpenTTD "scenario" file and a series of "saved games" which serve as starting points for each activity. To use the activities, each user must download the scenario file to their OpenTTD scenario folder (in Linux, this is usually in the $HOME/.openttd/scenario/ folder).

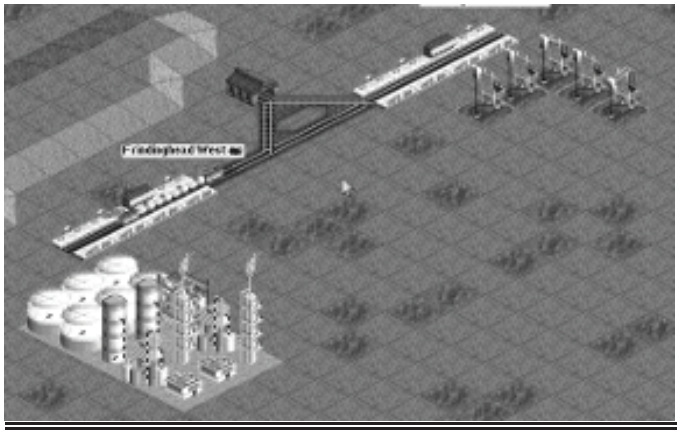

Figure 1: Track between an oil well and a refinery

The most common concurrency problems discussed in a systems class are race conditions, deadlocks, and starvation. Race conditions occur when two threads can simultaneously modify a shared data structure. Shared access can lead to problems where the value of a set of operations depends on the order of execution of the threads rather than the ordinary semantics of the operation. Deadlock occurs when two or more threads arrive at a situation in which each thread needs a resource held by another thread in order to make progress and therefore no thread can complete and release its resources. Starvation occurs when a set of threads prevents another thread from getting a needed resource, usually by repeatedly accessing the resource the starved thread needs in such a way that it can never complete.

USING OPENTTD

OpenTTD is an open-source game based on Transport Tycoon Deluxe by Microprose[12]. The game revolves around the construction of roads, railways, airports, and shipping routes to transport goods from the industries that produce them to the industries or cities that consume them. The construction of railways is an especially important element of the game. To transport goods from one industry to another, the player creates train stations at each endpoint and connects them to each other with track.

The player then adds a train depot to the track and uses the depot to create a train. Once the player has configured the train's destinations and way-points, it will navigate the track, picking up goods and delivering them.

Each delivery earns the player a reward, which he can use to purchase additional buildings, vehicles, or miles of track. The amount of the reward depends both on the delivery time and on the distance traveled. A more efficiently organized track leads to bigger rewards. A track in which deadlocks or starvation can occur can cause the player to lose resources very quickly. Figure 1 shows an example train track which connects the oil wells at the top right of the screen shot to the refinery at the bottom left. The building above the middle of the track is a depot which can create and service trains.

Tracks can contain many branches, loops, or terminals. To build a profitable rail empire, the player has to carefully arrange his tracks to maximize efficiency. In order to avoid collisions and deadlocks, the player can place signals along the track. There are two types of signals[13] in the game: block signals prevent a train from entering a section of track which is already occupied by another train, while path signals allow multiple trains to enter the same section of track, but only if both trains can reserve a safe path all the way through the section. Signals can be used to force trains to move in one direction around a circular track, to prevent multiple trains from accessing the same block of track at the same time, or to prevent collisions at intersections of track.

Signals and tracks can also be removed using the "bulldozer" tool. To correct an error in the track, the player first selects the tool used to create the track or signal, clicks on the bulldozer icon, and then clicks on the game object to be removed.

Creating safe OpenTTD tracks is very similar to creating thread-safe code. In both cases, the user manages a shared resource through the use of synchronization constructs. In both cases, an incorrect solution can lead either to a collision or a deadlock. In the train simulation, we address these problems using block and path signals. In programming, we typically use semaphores., which we define to be an integer valued data structure that provides the atomic operations "wait" and "signal". The "wait" operation decrements the semaphore, putting the current thread to sleep if the value becomes negative. The "signal" operation increments the semaphore, waking a sleeping thread if one exists.

RACE CONDITIONS

A race condition occurs when two threads write to a shared variable concurrently. When this happens, the value of the variable can depend on the order in which the threads execute. A block of code in which a race condition can occur is called a critical section. Unsynchronized access to a critical section can lead to incorrect program behavior (and sometimes even cause a crash), but might not manifest in any easily detectable way. The problem of preventing simultaneous access to a critical section is called "mutual exclusion".

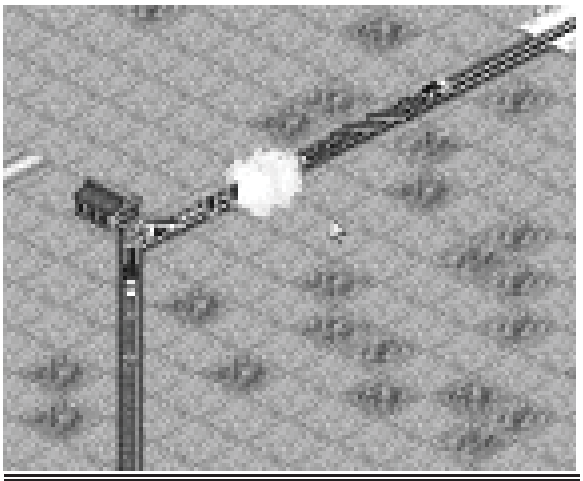

Figure 2: Trains collide in a critical section

In OpenTTD, race conditions occur when two trains share the same section of track. This section of track represents the "critical section". Often, nothing untoward will occur when this happens, especially if one train is following another. But if one train is slightly faster than the other or if the trains travel in opposite directions, simultaneous access can lead to a crash.

Activity one presents students with a scenario in which three trains share a critical section and asks them to use "block signals" as semaphores to prevent the trains from crashing. When the student loads the scenario and unchecks the pause button, the top two trains narrowly miss each other, which demonstrates both the danger of a race condition and that critical section violations do not always produce a detectable error. If the simulation continues to run for a few seconds, one of the trains will continue around the track and collide with the third train as shown in Figure 2. Students can prevent this problem by using block signals to protect the areas of the track which are shared by more than one train.

Simulating a race condition in OpenTTD is actually fairly difficult because the game AI will keep a train from leaving a station if it can't find a safe path to its first way point. To overcome this, we created a scenario in which the trains have already left the station and are already in a critical section. To do this, we deleted a section of track between the first train and the station. This allowed the second train to leave the station. We then restored the deleted track, which created a state in which both trains were in motion inside the critical

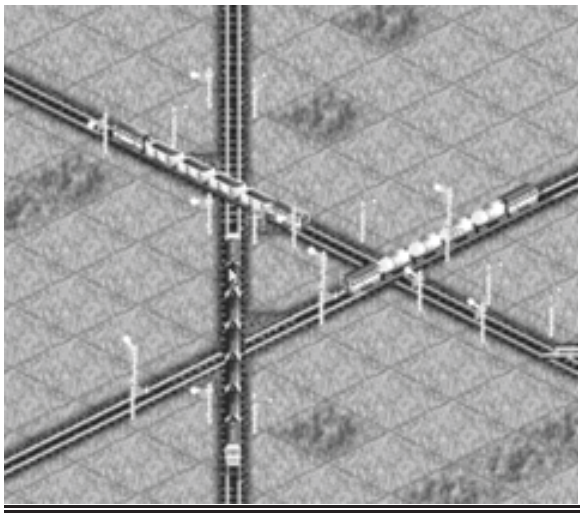

Figure 3: A three-way deadlock

section. Another approach might be to "flip" the activity, by setting up a thread-safe scenario and having the students remove signals to create the race condition.

DEADLOCKS

While semaphores are great for preventing race conditions, overzealous application can lead to another problem: deadlock. If a group of threads each hold a semaphore needed by the one of the other threads in the group, they can get stuck, causing the program to freeze.

Activity two presents students with a situation in which three trains are deadlocked at an intersection. Figure 3 shows the initial scenario. The first train is blocked by the second train. The second train is blocked by the third train. The third train is blocked by the first train.

To resolve this situation, the student has to back all the trains out of the intersection and then figure out which semaphores to remove (using the bulldozer tool) to solve the deadlock but preserve mutual exclusion of the critical section. One simple solution to the problem is to remove all the "internal" semaphores – the signals on the parts of the track which form a triangle – but leave the "external" semaphores which surround the triangle. This will force all the trains but one to stop outside the triangle in places which do not block the track. The first train to arrive will "get the semaphore" and proceed through the critical section. Once it has passed beyond the area of intersection, the other trains will proceed in order.

STARVATION

Even when a system is not deadlocked, it is possible for a thread to become blocked for long periods of time (or even forever) while other threads take turns monopolizing a resource it needs. This is called starvation.

Similarly, an OpenTTD train can experience starvation if a section of track which it needs access to passes back and forth between other, longer trains. This happens most often when trains can reserve huge blocks of track for long periods of time. Splitting the track up into smaller pieces usually solves this problem.

The third activity gives students a scenario in which one train starves while two other trains occupy the track it needs to escape from the depot. The student can solve this problem by splitting the track into smaller pieces that allow the third train time to "capture" enough track to escape.

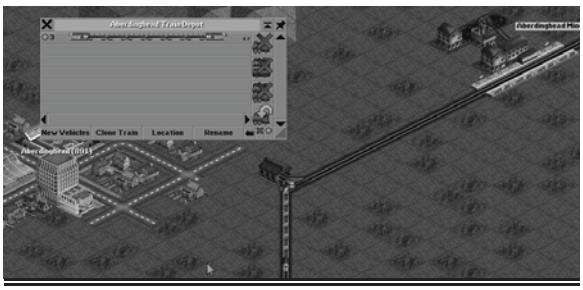

Figure 4: A train starves in the station

Figure 4 shows the starved train stuck in the Aberdinghead station while another train is using the track in front of the station. If the player waits long enough, the blocking trains will eventually break down or accumulate

enough of a delay from waiting at signals that the trapped train can acquire the track. In the meantime, however, the train consumes resources, but cannot leave the station.

To solve this problem, the student can break the track up into smaller segments. This will allow the trapped train to acquire the section of track immediately in front of the station whenever both of the trapping trains are in other segments. However, this solution has limited scalability – as the number of trains increases, the track becomes more and more crowded. Eventually, no subdivision of the track will be sufficient to prevent starvation, because the subdivisions will have to be so small that trains no longer fit into a single section of track. This problem, in turn, can be addressed by extending the track.

CONCLUSION

These projects are designed to supplement existing material in a course. It takes most students a little less than an hour to work through all three activities, which fits nicely into a 50-minute laboratory session.

We used these activities in an upper-level operating systems class at Longwood University, a small public liberal arts college. The assignment consisted of three broad stages. In the first stage, we introduced students to the OpenTTD user interface and gave them an overview of the relevant features of the game. This took about half of the class period. In the second stage, the students completed the three activities and experimented with the signal system. This discovery stage took most of the remaining time. In the third stage, students reflected on the activity by answering a set of questions about the project. For most students, this took only a few minutes.

Here are a few sample questions appropriate for the reflection section:

1. In order to prevent a race between two trains, you added signals to your track. Are there any drawbacks to adding signals? Can there be too many signals on the track?
2. At the end of the project, you should have a working train system, but if you pay close attention, you'll notice that the wood train spends a lot of time waiting. Experiment with adding and removing signals to see if you can make this system more efficient. What solutions did you find?
3. Suppose a multi-threaded program contains a shared data structure. If two threads try to write to the data structure simultaneously, a race condition can occur in which one thread overwrites the value of the other thread. Explain how this relates to what you did in your lab.

All of the students completed the assignment successfully. Students were very receptive to the activity and we received lots of positive feedback about how well the project helped them understand synchronization. In fact, several of them continued playing with the train scenario well after completing the checkpoints on which they were graded. The exercise worked well as an introduction to threads and synchronization and motivated further discussion of semaphore semantics and common semaphore patterns in class.

It is important to note that concurrency in OpenTTD has some subtle, but important differences from concurrency in systems programming. Probably the most significant difference is that race conditions have to be triggered by hand to circumvent the game's automatic crash protection. Another distinction is that, since all trains have the same behavior and set of operations, it is difficult to model asymmetric problems such as the "readers/writers" problem or a "producer/consumer" problem. Since OpenTTD is open source, it might be possible to extend the game to address these issues. It might also be possible to simulate these applications using more complicated tracks design (perhaps designating trains moving in one direction of the track as "readers" and trains moving the opposite direction as "writers"). However, since OpenTTD signals are essentially binary semaphores and the game doesn't have an equivalent to an integer-valued variable, it would be difficult to illustrate the traditional solutions to these problems without extensive modifications to the game engine.

ACKNOWLEDGEMENTS

OpenTTD was created by a large community of authors and artists. I am grateful for their work and especially appreciative for the wonderful documentation on the OpenTTD wiki[12,13]. The screenshots used in this paper are taken from OpenTTD 1.3.3. I am also grateful to Dr. Donald Blaheta who first suggested that this project might be worth sharing.

WORK CITED

[1] Akimoto, N., de Cheng, J., An educational game for teaching and learning concurrency, *Proceedings of the 1st international conference on knowledge economy and development of science and technology*, 34-39, 2003.

[2] Bedy, M., Carr, S., Huang, X., Shene, C., A visualization system for multithreaded programming, *SIGCSE Bulletin,* 32, (1), 1-5, 2000.

[3] Bi, Y., Beidler, J., A visual tool for teaching multithreading in Java, *Journal of Computing Sciences in Colleges,* 22, (6), 156-163, 2007.

[4] Burns, A., Davies, G., Pascal-FC: A language for teaching concurrent programming, *SIGPLAN Notices*, 23, (1), 58-66, 1988.

[5] Bynum, W. L., Camp. T., After you, Alfonse: a mutual exclusion toolkit, *SIGCSE Bulletin*, 28, (1), 170-174, 1996.

[6] Christopher, W. A., Procter, S. J., Anderson, T.E., The Nachos instructional operating system, Proceedings of the USENIX Winter Conference, 481-490, 1993.

[7] Donaldson, J. L., Teaching operating systems in a virtual machine environment, *SIGCSE Bulletin*, 19, (1), 206-211, 1987.

[8] Higginbotham, C. W., Morelli, R., A system for teaching concurrent programming, *Proceedings of the Twenty-Second SIGCSE Technical Symposium on Computer Science Education*, 309-316, 1991.

[9] Holland, D.A., Lim, A. T., Seltzer, M.I., A new instructional operating system, *Proceedings of the 33rd SIGCSE technical symposium on computer science education,* 111-115, 2002.

[10] Hoskey, A., Simple OS: A component-based operating system simulator in the spirit of the little man, *Journal of Computing Sciences in Colleges,* 117-124, 2013

[11] Kolikant, Y. B., Ben-Ari, M., Pollack, S., The anthropology semaphores, *Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE conference on innovation and technology in computer science education*, 21-24, 2000.

[12] OpenTTD developers, OpenTTD Wiki, http://wiki.openttd.org, retrieved April 2014.

[13] OpenTTD developers, OpenTTD Wiki, http://wiki.openttd.org/signals/, retrieved April 2014.

[14] Pfaff, B., Romano, A., Back, G., The pintos instructional operating system kernel, *Proceedings of the 40th ACM technical symposium on computer science education*, 453-457, 2009.

[15] Reese, D., Using multiplayer games to teach interprocess communication mechanisms, *SIGCSE Bulletin*, 32, (4), 45-47, 2000.